Here's the thing: computers are built on **layers of abstraction.** Each layer, from hardware, firmware, operating system, programming language, and application, hides the complexity beneath it. When you write code or use a website, you don't think about transistors or voltage levels or how the CPU decodes instructions. You just call functions. The layers beneath handle the rest.

In this article, I'll focus on Storage and Processing, the parts that happen inside the computer and remain mysterious to most of us. The computer turns your input actions into signals and instructions, and the result is something you can interact with. The real question is where those signals and instructions live when they're inside, and how the computer transforms them? We'll unravel all of that.

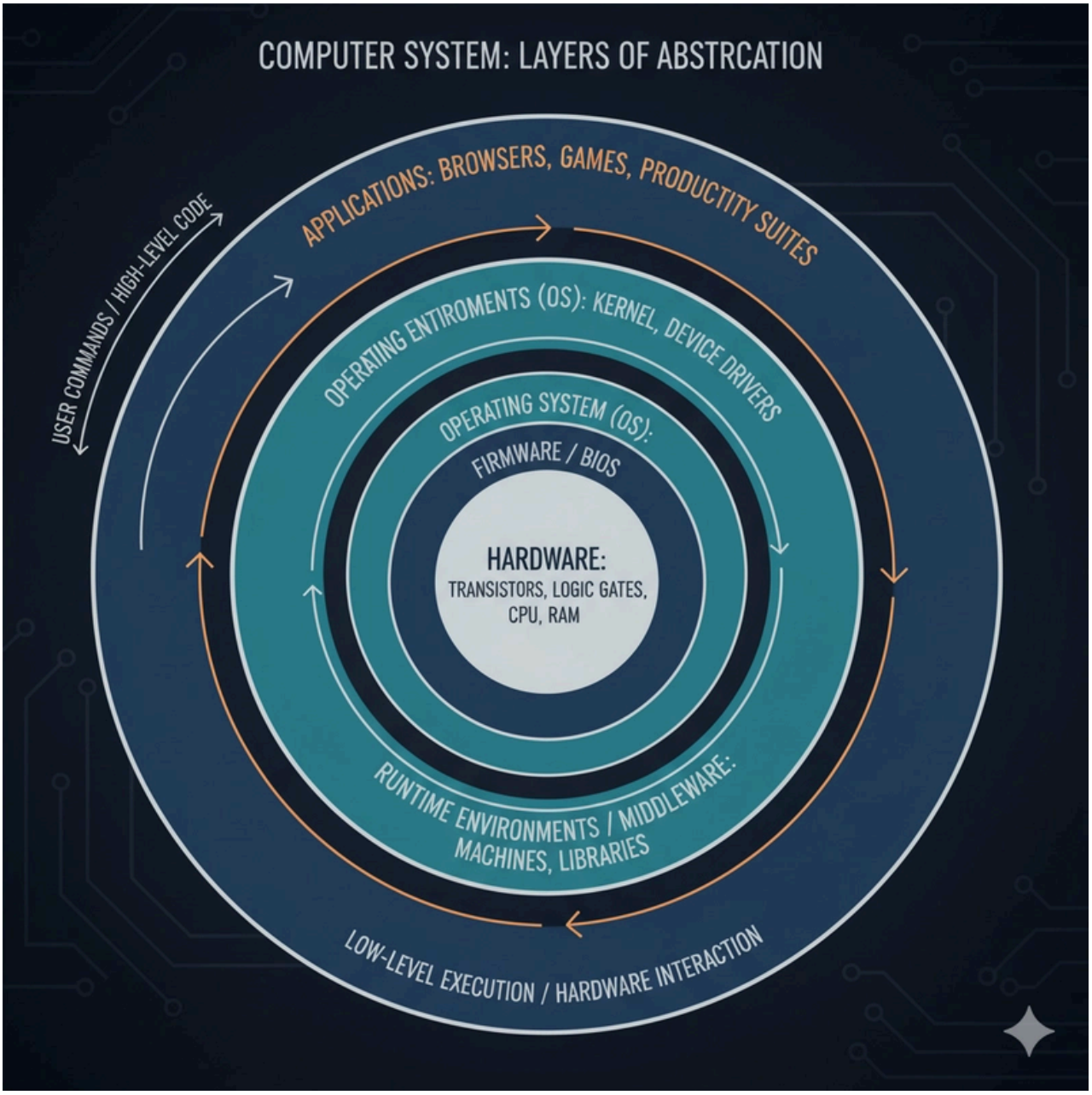

Image: *A concentric ring or pyramid diagram showing the layer of abstraction progression from* ***Hardware*** *→* ***OS*** *→* ***Applications****.*

The Language: Why Binary and How Bits Encode Everything

Computers don't see the world the way we do. At the most basic level, they only speak binary: a language of 1s and 0s.💻 Each 1 or 0 is called a **bit** (short for binary digit), and it's the fundamental unit of information. This means that every input or output, including every photo, song, video, email, and program you've ever interacted with, gets translated into or from binary data.

Modern computers are digital. They don't just relay information. They give electrical signals meaning. A signal under a certain voltage threshold is interpreted as "off" (0), and over a certain threshold is "on" (1). The continuous voltage range is reduced to just two discrete states. This is where the term "digital" originates: from the digits 0 and 1.

You might wonder why computers use only two values (0 and 1) instead of ten, like our decimal number system (0 to 9). The answer is that two states are much easier to tell apart reliably. When you're working with electrical signals, noise and variations are inevitable. With just two distinct states, one representing 0 and the other representing 1, there's no ambiguity, even when conditions aren't perfect. More states would mean more confusion, more errors, more complexity.

Binary works just like decimal, except with powers of two. For example, 01001000 in binary equals 72 in decimal. This value represents the letter 'H' in ASCII, a standard whose first 128 characters are directly adopted by Unicode.

With 8 bits (a byte), you can represent 256 possible values (2^8 = 256, ranging from 0 to 255). Combine enough bits, and you can represent anything: numbers, letters, images, sound, video. What makes that possible are standards. The computer doesn't "know" what 'H' is. It just follows agreed-upon rules like Unicode for text, RGB for images, or WAV for sound.

Now, since everything boils down to numbers, how do we store something like a song or a video? Through encoding standards like MP4 for video or JPEG for images, these get converted to numbers that the computer can store and manipulate.

Everything inside the computer, everything Storage holds, and everything Processing touches, is built on this foundation of 1s and 0s. But those bits are not abstract ideas. They're physical voltages controlled by billions of switches working together. Let's see how

The Hardware: From Silicon to Logic

Now let's descend to the hardware layer and talk about how the computer physically creates those 1s and 0s using electricity, and how it uses them to make decisions and perform operations. Remember those binary digits we just discussed? Here's how they physically exist inside the machine.

The Motherboard

Inside your computer case, you'll find many things: a power supply, cooling fans, storage drives, and various cables connecting everything together. But **most of the action takes place on the motherboard**—the main circuit board of the computer, so that's where we'll focus.

The motherboard is a printed circuit board (PCB), a flat board with copper wires etched into it, connecting various components. The board can have multiple layers for efficient signal routing.

The key innovation for modern computers is the integrated circuit (IC), commonly called a chip or microchip. An IC is essentially a complete electronic circuit etched onto a single piece of silicon. This allows billions of tiny components called transistors to be packed onto a chip smaller than your fingernail.

Key components on the motherboard:

- **The CPU (Central Processing Unit)** - A single chip that plugs into a socket on the motherboard. This is where all the processing happens. Inside the CPU are several specialized components working together: a **control unit** that orchestrates everything, an **ALU** that performs calculations, tiny fast memory locations called **registers**, and small pools of ultra-fast memory called **cache**.
- **RAM (Random Access Memory)** - Memory sticks that slot into the motherboard. This is the fast, temporary storage for data that the computer is actively working on.
- **Chipset** - A set of chips that manage data flow between the CPU, memory, and peripheral devices. Think of it as the motherboard's traffic controller, coordinating communication between all the major components.
- **BIOS/UEFI chip** - Stores firmware (the low-level software layer between hardware and the operating system) that runs when you first turn on the computer. This firmware performs hardware checks and loads the operating system from storage.
- **Ports** - USB ports, network ports, etc., that connect external devices.

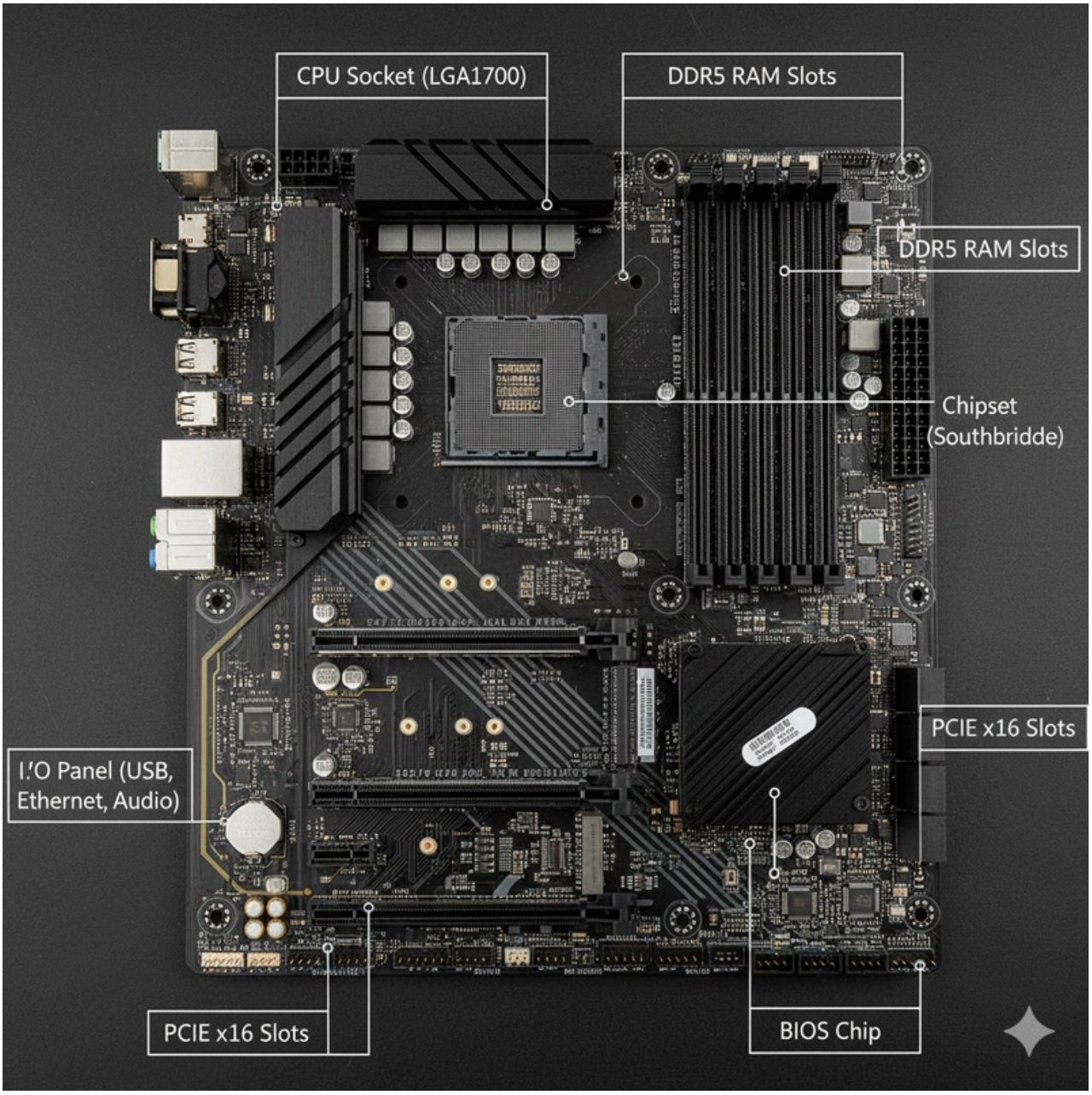

***Image**:Motherboard layout showing the main circuit board with key components like the CPU socket, RAM slots, chipset, and ports.*

We'll focus on what happens inside the CPU chip, because that's where processing happens.

Inside the CPU: Transistors and Gates

Under the hood, bits are voltages. In modern circuits, a "high" voltage encodes 1, a "low" voltage encodes 0. Numbers are literally voltage levels flowing through wires. The component responsible for controlling these voltages is called a transistor, an electrically controlled switch that's the fundamental building block of all digital electronics.

The type used in modern CPUs is called a MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor). Apply voltage to the control? Current flows. Switch ON. Remove voltage? Current stops. Switch OFF.

A modern processor can pack tens of billions of transistors into a chip the size of your fingernail. Apple's M3 chips, for example, contain over 90 billion transistors 🤯, each one flipping on and off billions of times per second.

Wiring these transistors in specific patterns creates **logic gates**. Gates are tiny circuits that implement rules. They take voltage inputs (1s and 0s) and produce voltage outputs based on logical operations, like AND gates that only output 1 when both inputs are 1, or NOT gates that flip their input.

Combine enough of these gates, and you can build something far greater: the **ALU (Arithmetic Logic Unit)**—the part of your CPU that adds, subtracts, compares numbers, and performs logical operations. All computer math happens here. Want to add two numbers? The ALU chains together XOR gates and AND gates in a specific pattern called an adder circuit. But there's a subtle problem lurking here.

The Clock: Keeping Everything in Sync

Here's something interesting about voltage. It takes time to ripple through gates. Send a signal through one gate, and it might take a nanosecond to stabilize. Chain thousands of gates together and you've got signals arriving at different times, some still changing while others have already settled. Without coordination, it's chaos. Results would be corrupted mid-flight. Outputs would be garbage.

That's where the **clock** comes in. Think of the clock like an orchestra conductor 🎵 keeping all the musicians in time. A CPU's clock ticks billions of times per second (measured in GHz, a 3.5 GHz CPU ticks 3.5 billion times per second). The clock generates consistent electrical pulses sent down the wires. It does this the same way a digital watch keeps time: using a quartz crystal that vibrates at a precise frequency when electricity is applied.

Between ticks: Voltages propagate through gates. Computation is happening, signals are flowing, and math is being done.

On each tick: Results get locked into place (stored in small memory circuits) so the next stage can start fresh with stable values.

The clock remains the fundamental heartbeat that synchronizes everything and keeps electrons marching in formation.

So now we have transistors that can switch on and off to represent 1s and 0s. We have gates that use those switches to make decisions. We have an ALU that chains gates together to do math. And we have a clock that keeps everything synchronized. That's processing. But processing is useless without somewhere to put the data.

Memory: Where Data Lives

Moving up a layer, we need to understand where data lives. Processing is only half the story. The CPU needs somewhere to store the data it's working on, and not all storage is created equal. Modern computers use a memory hierarchy, balancing trade-offs between speed, size, and cost. Think of memory like a pyramid:

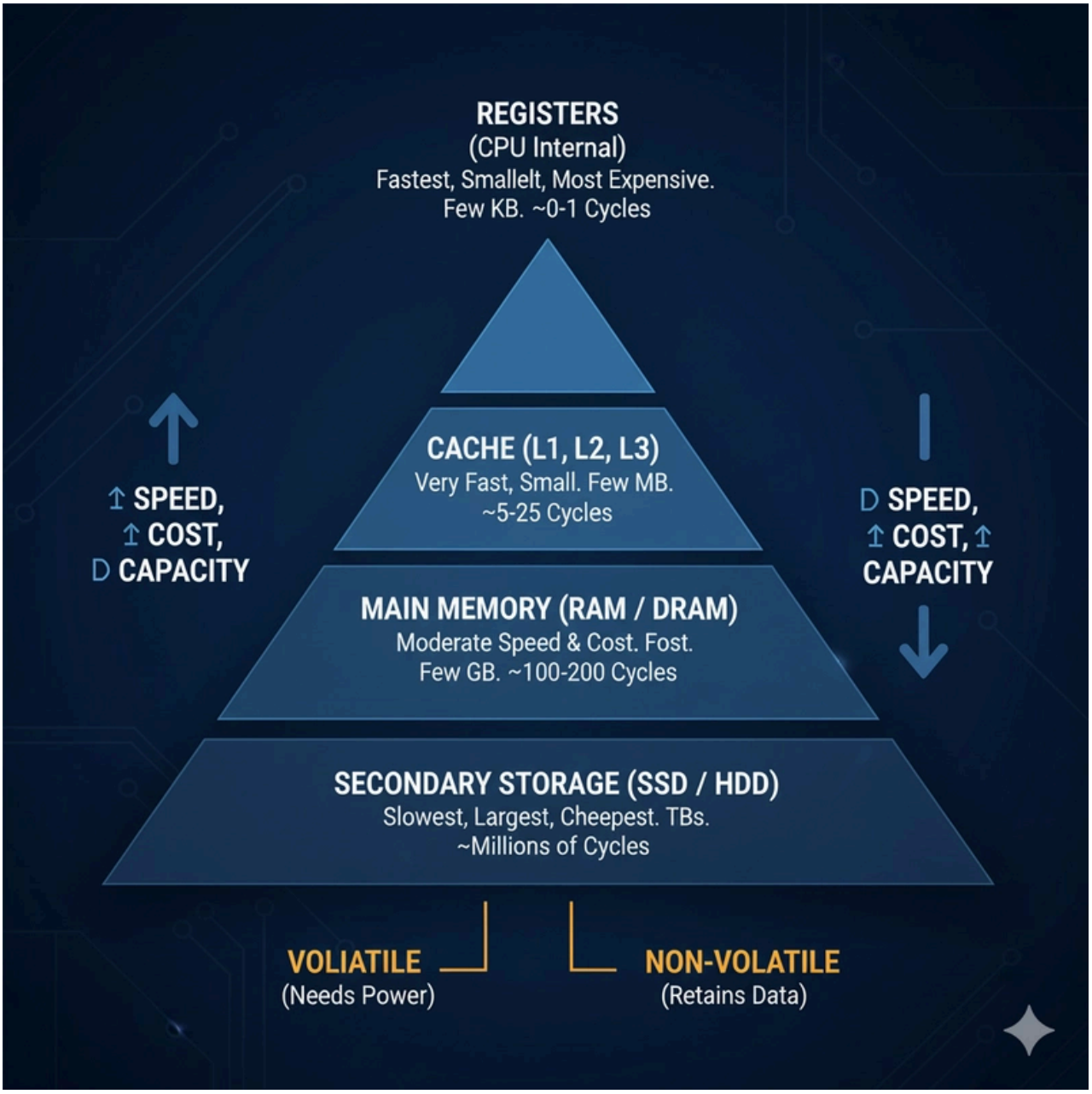

***Image**: A pyramid diagram illustrating the **memory hierarchy**, showing the trade-off in speed, size, and cost. It progresses from the fastest layer **Registers** at the top, through **Cache** L1/L2/L3 and **RAM**, to **Storage** the largest layer at the bottom.*

At the top (fastest, smallest, most expensive):

- **Registers** - Tiny storage locations inside the CPU itself. A CPU might have only a few dozen registers, each holding a single number (typically 64 bits). These are where the CPU keeps the data it's using *right now*. There are general-purpose registers for holding any data, and special-purpose registers like the **program counter** which tracks which instruction to execute next and the **instruction register** which holds the current instruction being processed.
- **Cache** - Small, fast memory also very close to the CPU. Split into levels: L1 (smallest, fastest, typically per-core), L2 (bigger, slower, also per-core in modern CPUs), L3 (bigger still, slower still, usually shared across all cores). Modern CPUs might have 256 KB of L1 per core, 1-8 MB of L2 per core, and 8-64 MB of shared L3.

In the middle:

RAM (Random Access Memory) - The main working memory. Your computer might have 8, 16, or 32 GB of RAM. Much larger than cache, but slower to access.

At the bottom (slowest, largest, cheapest):

Storage - SSDs and hard drives. Terabytes of space, but thousands of times slower than RAM.

This hierarchy exists because of physics and economics. You can't make infinite amounts of super-fast memory, it's expensive and generates too much heat. So computers keep a little bit of ultra-fast memory close to the CPU and larger pools of slower memory further away. But how do these different types of memory actually work, and how does data move between them?

Connecting CPU and RAM: Buses

Before I show you how memory works, remember that data doesn't magically move; it travels along wires grouped into **buses** (they "transport" information, like a bus transports passengers).

There are two critical buses connecting the CPU and RAM:

- **Address bus**: The CPU sends memory addresses along this bus to tell RAM which location it wants to access. Think of it like telling a librarian which book you want by giving them the call number.
- **Data bus**: The actual data travels back and forth along this bus. This is typically 64 bits wide (64 wires running in parallel), allowing 64 bits to transfer simultaneously.

When the CPU needs data from RAM, it sends the address on the address bus, and RAM responds by sending the data back on the data bus. This communication takes time, hundreds of clock cycles. This is why the memory hierarchy exists: to keep frequently used data closer.

RAM: The Active Workspace

The most common type of RAM is **DRAM** (Dynamic RAM). Each bit is stored as a tiny electric charge in a capacitor. If the capacitor is charged, that's a 1. If it's empty, that's a 0.

Reading the charge tells you the bit's value, but reading is destructive, it discharges the capacitor, which must be recharged. Additionally, capacitors naturally leak charge over time, so RAM constantly refreshes itself thousands of times per second. This is why it's called *Dynamic* RAM.

Here's the catch: DRAM is volatile memory because it requires constant power to maintain data. When you turn off the power, the capacitors lose their charge, and everything in RAM disappears. That's why you have to save your work to disk which uses non-volatile memory that retains data without power, like SSDs using flash memory or traditional hard drives using magnetic storage.

Cache and registers use a different technology: SRAM (Static RAM). Instead of capacitors, SRAM uses circuits made from transistors wired in a configuration that holds its state as long as power flows, no constant refreshing is needed. The same transistors that make logic gates can also be configured to store values. This makes SRAM much faster than DRAM, but also much more expensive and physically larger for the same amount of storage. That's why you only get megabytes of cache versus gigabytes of RAM.

Persistent Storage: Data That Lasts

RAM is fast but forgets everything when you power off. For data that needs to survive, computers use persistent storage like hard drives and SSDs.

Hard Disk Drives (HDDs) store data using magnetism. Inside are spinning metal platters with a tiny head floating above them like a record player. The head magnetizes microscopic spots on the disk, one direction is 1, the opposite is 0. Reading just detects those magnetic fields.

The problem? Moving parts. The head has to physically swing to the right location, and the platter has to spin to the right spot. That takes time. Millions of clock cycles. But HDDs are cheap and can store terabytes.

SSDs use flash memory, a special type of transistor with an extra "floating gate" that can trap electrons. Force electrons onto that gate, and they stay there even without power. That's your 1. No electrons? That's 0. Reading just checks if charge is trapped.

No moving parts means SSDs are significantly faster than HDDs. They're also more expensive per gigabyte, but for speed, nothing beats them.

Modern computers often use both: SSD for the OS and programs, HDD for bulk storage.

Why Location Matters

Here's a quick example: Let's say the CPU needs to add two numbers.

1. **If the numbers are in registers:** 1-2 clock cycles. Nearly instant.⚡
2. **If they're in L1 cache:** 4 cycles.
3. **If they're in L3 cache:** 40 cycles.
4. **If they're in RAM:** 200 cycles.
5. **If they're on disk:** Millions of cycles.🐌

This is why good software always tries to keep frequently-used data "close" to the CPU. This principle is called **locality of reference**. If the CPU just used a piece of data, it'll probably need it again soon, so keep it in cache. Modern CPUs are smart about this. They automatically predict what data you'll need next and pre-load it into cache. When they guess right, your program is faster. When they guess wrong, you wait.

Now here's where it all comes together. We've got transistors and gates that can compute. We've got memory that can store data at different speeds. But how does the CPU orchestrate all of these to actually run a program?

Processing: How the CPU Executes Instructions

We've seen the building blocks—transistors, gates, and memory. Now, let's explore how the CPU actually *runs* a program.

The Instruction Cycle

Every program is a list of instructions stored in RAM. Each instruction is just a number; some bits describe the operation (add, subtract, load, store), and other bits point to where the data lives in memory. The CPU runs these instructions one at a time in a loop called the **instruction cycle** also called the fetch-decode-execute cycle.

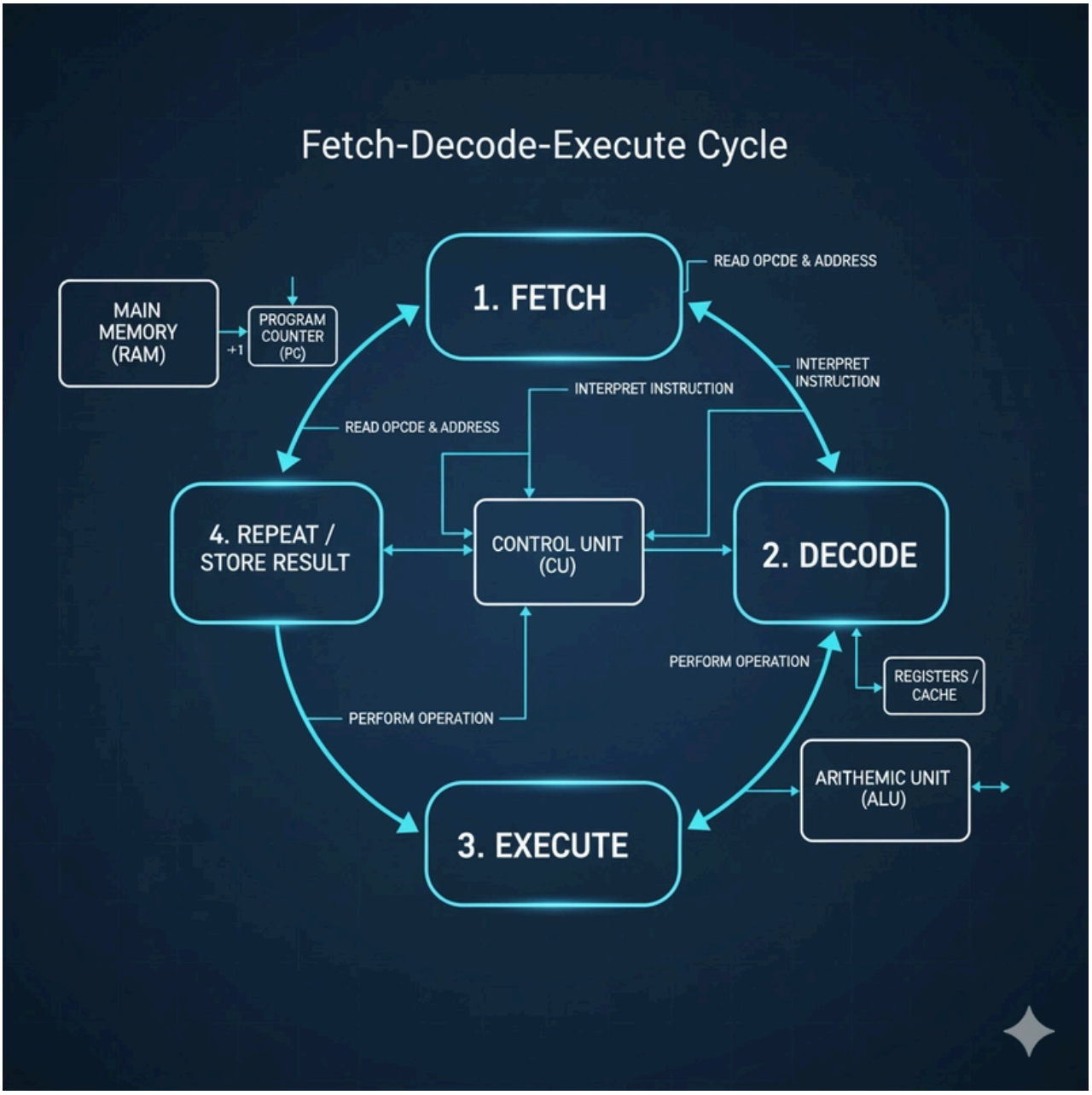

Image: *A flow visualization of the* ***fetch-decode-execute cycle****, showing how instructions flow from* ***RAM*** *to the* ***instruction register****, through the* ***control unit*** *and* ***ALU/registers****, and back to memory.*

It's worth noting that different CPUs use different instruction sets (called ISAs)—x86-64, ARM, RISC-V, etc. Each has its own binary encoding, which is why software compiled for one processor won't run on another without translation.

Putting It All Together

Here's a simple example. Imagine a program that adds two numbers and stores the result. In assembly language (a human-readable version of machine code), it'll look like this:👇

```
LOAD R1, [1000] ; Load the value at memory address 1000 into register R1LOAD R2,
[1004] ; Load the value at memory address 1004 into register
R2ADD R3, R1, R2 ; Add R1 and R2, store result in R3STORE
R3, [1008] ; Store R3's value back to memory address 1008
```

What the CPU does:

1. **Fetch** the first instruction from RAM
2. **Decode** it: "LOAD means get data from memory"
3. **Execute**: Send address 1000 to RAM, get back the value, and put it in register R1
4. Move to the next instruction, repeat for R2
5. **Fetch** the ADD instruction
6. **Decode**: "ADD means use the ALU"
7. **Execute**: The ALU takes values from R1 and R2, adds them using XOR and AND gate circuits, and puts the result in R3
8. **Fetch** the STORE instruction
9. **Execute**: Send R3's value to RAM at address 1008

Each step is just voltages changing, gates opening and closing, electrons flowing through silicon. But layer enough of these steps together, and you get everything a computer can do.

Modern CPUs do clever tricks to go faster: they fetch the next instruction while decoding the current one (called **pipelining**), they execute multiple instructions at once on different parts of the chip (called **superscalar execution**), and they run multiple programs simultaneously on different **cores** (multiple CPUs on one chip). But the fundamental cycle —fetch, decode, execute—is always there, happening millions of times in the blink of an eye.

The Conductor: How the OS Brings It All Together

We've covered most of storage and processing, the core of how computers work. But there's one more piece needed to make this all useful: something has to coordinate everything we've discussed.

That's where the **operating system** comes in. Think of it as the conductor of the orchestra we've been building.

You're probably running dozens of programs right now. Your browser, music player, and text editor, they're all loaded in RAM, all waiting for CPU time. The OS manages this by giving each program a tiny slice of CPU time, maybe 10 milliseconds, then switching to the next program. It happens so fast you don't notice. This is **multitasking** in play.🔄

The OS also protects programs from each other. Each program thinks it has full access to RAM, but the OS carves up memory into protected sections. If one program crashes, it can't corrupt another's data. A piece of hardware called the MMU (Memory Management Unit) works with the OS to implement virtual memory, where each program gets its own

isolated address space. The MMU translates these virtual addresses to actual physical RAM addresses. When RAM fills up, the OS can even move unused data to disk (called paging or swapping), making programs think there's more RAM than physically exists, though accessing this disk-backed memory is much slower than real RAM as we now know.

When your program needs to talk to hardware—display something on screen, save a file, or send data over the network—it doesn't do it directly. It asks the OS through a **system call**, and the OS handles the details using **device drivers** that know how to communicate with specific hardware.

This layering means you can write code that says "save this file" without knowing anything about how your SSD controller works. The OS translates your high-level request into the low-level commands that storage hardware understands.

Conclusion

So, how do computers actually work?

At the bottom, billions of transistors switch on and off, representing 1s and 0s as voltage levels. Move one layer up, and logic gates combine those 1s and 0s into decisions, building up to ALUs that perform math using adder circuits and flags. Another layer up, the memory hierarchy balances speed and size—registers, cache, RAM, and storage each serve data at different speeds, connected by buses. Another layer up, the CPU executes its cycle: fetch, decode, execute, all synchronized by a clock ticking thousands of times per millisecond. One more layer, and the OS orchestrates it all, managing programs, protecting memory, and coordinating hardware. At the top sits your code, blissfully unaware of the intricate dance playing out below.

Remember when I said the gap between typing a command and pixels lighting up was a total mystery? Now you know: your keypress becomes a voltage signal, travels through buses, gets stored as charges in RAM, triggers CPU instructions fetched from memory, executed by billions of transistors arranged in logic gates, all synchronized by a clock ticking billions of times per second, with the result written back to memory and sent to your graphics card to light up specific pixels. What once seemed like magic is now just... Well, really pretty clever engineering.✨

Does understanding all this make you a better programmer? Maybe not directly. But there's something deeply satisfying about knowing how these various components make up the computer. And that, I think, is pretty remarkable.

Here’s a list of amazing resources I’ve found while doing some research for this post:🙂. 📚

Crash Course Computer Science: Episodes 5, 7 and 17 - Good resource on CPU operations, memory hierarchy, and instruction cycles